Poll Everywhere

pollev.com/cse333

About how long did Exercise 3 take you?

A. **[0, 2) hours**
B. **[2, 4) hours**
C. **[4, 6) hours**
D. **[6, 8) hours**
E. **8+ Hours**
F. **I didn't submit / I prefer not to say**

UNIVERSITY *of* WASHINGTON

File I/O: Cstdio, Buffering, POSIX
CSE 333 Winter 2023

Instructor: Justin Hsia

Teaching Assistants:

Adina Tung	Danny Agustinus	Edward Zhang
James Froelich	Lahari Nidadavolu	Mitchell Levy
Noa Ferman	Patrick Ho	Paul Han
Saket Gollapudi	Sara Deutscher	Tim Mandzyuk
Timmy Yang	Wei Wu	Yiqing Wang
Zhuochun Liu		

Relevant Course Information

❖ Exercise 4 posted today, due *next Thursday* (1/26)

- Longest exercise and HW1 is due; Exercise 5 will be released on Monday, due Friday (1/27)

❖ Homework 1 due Thursday night (1/19)

- Clean up "to do" comments, but leave "STEP #" markers
- Graded not just on correctness, also code quality
- OH get crowded – come prepared to describe your incorrect behavior and what you think the issue is and what you've tried
- Late days: don't tag `hw1-final` until you are really ready
  - Please use them if you need to!

❖ Homework 2 released on Friday

- Partner declaration form and matching form will be released later today

3

Lecture Outline

- ❖ **File I/O with the C standard library**
- ❖ C Stream Buffering
- ❖ POSIX Lower-Level I/O

File I/O

- ❖ We'll start by using C's standard library
  - These functions are part of `glibc` on Linux
  - They are implemented using Linux system calls (POSIX)

- ❖ C's `stdio` defines the notion of a stream
  - A sequence of characters that flows **to** and **from** a device
    - Can be either *text* or *binary*; Linux does not distinguish
  - Is *buffered* by default; `libc` reads ahead of your program *→ console (un buffered)*
  - Three streams provided by default: `stdin`, `stdout`, `stderr` *keyboard → console* *console (buffered)*
    - You can open additional streams to read and write to files
  - C streams are manipulated with a `FILE*` pointer, which is defined in `stdio.h`

C Stream Functions (1 of 2)

❖ Some stream functions (complete list in `stdio.h`):

NULL if error!

■ `FILE* fopen(filename, mode);`

• Opens a stream to the specified file in specified file access mode

■ `int fclose(stream);`

• Closes the specified stream (and file)

■ `int fprintf(stream, format, ...);`

• Writes a formatted C string
  – `printf(...);` is equivalent to `fprintf(stdout, ...);`

■ `int fscanf(stream, format, ...);`

• Reads data and stores data matching the format string

C Stream Functions (2 of 2)

❖ Some stream functions (complete list in `stdio.h`):

- `FILE* `**`fopen`**`(filename, mode);`
  - Opens a stream to the specified file in specified file access mode

- `int `**`fclose`**`(stream);`
  - Closes the specified stream (and file)

- `size_t `**`fwrite`**`(ptr, size, count, stream);`

  tries to move size * count bytes total

  - Writes an array of *count* elements of *size* bytes from *ptr* to *stream*

  # of elements actually moved (unsigned)

- `size_t `**`fread`**`(ptr, size, count, stream);`
  - Reads an array of *count* elements of *size* bytes from *stream* to *ptr*

  ptr → *user array*

C Stream Error Checking/Handling

❖ Some error functions (complete list in `stdio.h`):

- `int` **`ferror`**`(stream);`
  - Checks if the error indicator associated with the specified stream is set

- `int` **`clearerr`**`(stream);`
  - Resets error and EOF indicators for the specified stream

- `void` **`perror`**`(message);`
  - Prints `message` followed by an error message related to errno to `stderr`

global var → errno

extra info!

① `fprintf(stderr, ... );` // single, known cause

error message → ② `perror("my error msg");` // multiple possible causes

8

C Streams Example

cp_example.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#define READBUFSIZE 128

int main(int argc, char** argv) {
  FILE* fin;
  FILE* fout;              ← stream variables
  char readbuf[READBUFSIZE];  ← arbitrarily-sized buffer
  size_t readlen;
                         prints to console, even if you pipe program output
  if (argc != 3) {
    fprintf(stderr, "usage: ./cp_example infile outfile\n");
    return EXIT_FAILURE;         // defined in stdlib.h
  }

  // Open the input file      file must exist when reading
  fin = fopen(argv[1], "rb");  // "rb" -> read, binary mode
  if (fin == NULL) {
    perror("fopen for read failed");  ← prints extra info on source of error
    return EXIT_FAILURE;
  }
  ...    // next slide's code
```

9

C Streams Example

cp_example.c

```
int main(int argc, char** argv) {

  ...    // previous slide's code

  // Open the output file
  fout = fopen(argv[2], "wb");   // "wb" -> write, binary mode
  if (fout == NULL) {
    perror("fopen for write failed");
    fclose(fin);
    return EXIT_FAILURE;
  }

  // Read from the file, write to fout
  while ((readlen = fread(readbuf, 1, READBUFSIZE, fin)) > 0) {
    // Test to see if we encountered an error while reading
    if (ferror(fin)) {
      perror("fread failed");
      fclose(fin);
      fclose(fout);
      return EXIT_FAILURE;
    }
    ...    // next slide's code
}
```

Handwritten annotations:

when writing, file created if it doesn't exist

← make sure to clean up for every exit path!

#of bytes actually read

(check if error on input stream)

for file of size 300 bytes, fread called 4 times:

① readlen=128
② readlen = 128
③ readlen=44
④ readlen = 0

0   300

C Streams Example

cp_example.c

```c
int main(int argc, char** argv) {

  ...    // two slides ago's code

  ...    // previous slide's code

    if (fwrite(readbuf, 1, readlen, fout) < readlen) {
      perror("fwrite failed");
      fclose(fin);
      fclose(fout);
      return EXIT_FAILURE;
    }
  }

  fclose(fin);
  fclose(fout);

  return EXIT_SUCCESS;
}
```

(handwritten annotations) return value from fread

something wrong if didn't write all requested bytes

close streams when done with them!

Lecture Outline

- ❖ File I/O with the C standard library
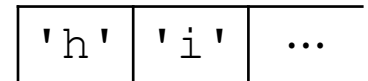- ❖ **C Stream Buffering**
- ❖ POSIX Lower-Level I/O

Buffering

❖ By default, `stdio` uses buffering for streams:

■ Data written by **fwrite**() is copied into a buffer allocated by `stdio` inside your process' address space

■ As some point, the buffer will be "drained" into the destination:
  • When you explicitly call **fflush**() on the stream
  • When the buffer size is exceeded (often 1024 or 4096 bytes)
  • For `stdout` to console, when a newline is written (*"line buffered"*) or when some other function tries to read from the console
  • When you call **fclose**() on the stream
  • When your process exits gracefully (**exit**() or `return` from **main**())
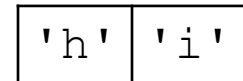
Buffering Example

```c
int main(int argc, char** argv) {
  FILE* fout = fopen("test.txt", "wb");


  // write "hi" one char at a time
  if (fwrite("h", sizeof(char), 1, fout) < 1) {
    perror("fwrite failed");
    fclose(fout);
    return EXIT_FAILURE;
  }

  if (fwrite("i", sizeof(char), 1, fout) < 1) {
    perror("fwrite failed");
    fclose(fout);
    return EXIT_FAILURE;
  }

  fclose(fout);
  return EXIT_SUCCESS;
}
```

buffered_hi.c

C stdio buffer

'h'	'i'	...

test.txt (disk)

'h'	'i'

No Buffering Example

```c
int main(int argc, char** argv) {
  FILE* fout = fopen("test.txt", "wb");
  setbuf(fout, NULL); // turn off buffering

  // write "hi" one char at a time
  if (fwrite("h", sizeof(char), 1, fout) < 1) {
    perror("fwrite failed");
    fclose(fout);
    return EXIT_FAILURE;
  }

  if (fwrite("i", sizeof(char), 1, fout) < 1) {
    perror("fwrite failed");
    fclose(fout);
    return EXIT_FAILURE;
  }

  fclose(fout);
  return EXIT_SUCCESS;
}
```
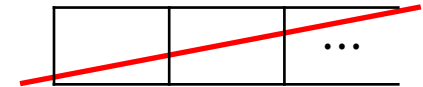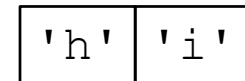
C stdio buffer

...

test.txt (disk)

'h'	'i'

unbuffered_hi.c

Why Buffer?

❖ Performance – avoid disk accesses

- Group many small writes into a single larger write

- Disk Latency = 😱 😱 😱 (Jeff Dean from LADIS '09)

orders of magnitude still accurate!

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

❖ Convenience – nicer API

- We'll compare C's **fread()** with POSIX's **read()**

Why NOT Buffer?

- ❖ Reliability – the buffer needs to be flushed
  - ▪ Loss of computer power = loss of data
  - ▪ "Completion" of a write (*i.e.*, return from **fwrite()**) does not mean the data has actually been written
    - • What if you signal another process to read the file you just wrote to?

- ❖ Performance – buffering takes time
  - ▪ Copying data into the `stdio` buffer consumes CPU cycles and memory bandwidth
  - ▪ Can potentially slow down high-performance applications, like a web server or database (*"zero-copy"*)

- ❖ When is buffering faster? Slower?

  many small writes / large writes

Lecture Outline

- ❖ File I/O with the C standard library
- ❖ C Stream Buffering
- ❖ **POSIX Lower-Level I/O**

From C to POSIX

❖ Most UNIX-en support a common set of lower-level file access APIs: POSIX – Portable Operating System Interface

▪ **open(), read(), write(), close(), lseek()**

  • Similar in spirit to their `f*()` counterparts from the C std lib

  • Lower-level and unbuffered compared to their counterparts

  • Also less convenient

▪ You will have to use these to read file system directories and for network I/O, so we might as well learn them now

  • These are functionalities that C stdio *doesn't* provide!

open/close

❖ To open a file:

- Pass in the filename and access mode (similar to **fopen**)

- Get back a "file descriptor"
  - Similar to `FILE*` from **fopen**, but is just an int
  - **−1** indicates an error

```c
#include <fcntl.h>     // for open()
#include <unistd.h>    // for close()
  ...                  filename        access mode
  int fd = open("foo.txt", O_RDONLY);
  if (fd == -1) {
    perror("open failed");
    exit(EXIT_FAILURE);
  }
  ...        file descriptor
  close(fd);
```
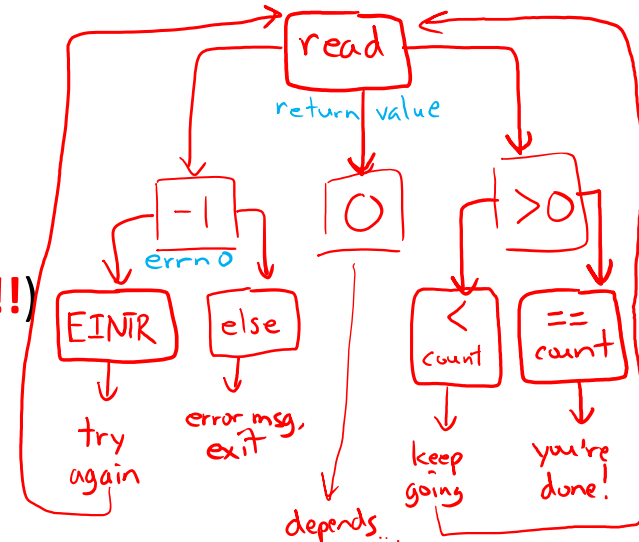
❖ Open descriptors: 0 (`stdin`), 1 (`stdout`), 2 (`stderr`)

Reading from a File

try to read count bytes

❖ ```
ssize_t read(int fd, void* buf, size_t count);
```

- Advances forward in the file by number of bytes read

- Returns the number of bytes read
  - Might be fewer bytes than you requested (**!!!**)
  - Returns **0** if you're already at the end-of-file
  - Returns **−1** on error (and sets `errno`)

*read*
*return value*
*−1* *errno* *EINTR* *else* *try again* *error msg, exit*
*0* *depends...*
*>0* *<count* *==count* *keep going* *you're done!*

- There are some surprising error modes (check `errno`)

*these are defined in*
*errno.h*

  - `EBADF`:    bad file descriptor
  - `EFAULT`:   output buffer is not a valid address
  - `EINTR`:    read was interrupted, please try again  (ARGH!!!! 😡 😠)
  - And many others…

# Poll Everywhere

**pollev.com/cse333**

# We want to read 'n' bytes. Which is the correct completion of the blank below?

```
char* buf = ...;   // buffer of size n
int bytes_left = n;
int result;        // result of read()

while (bytes_left > 0) {
  result = read(fd, _____, bytes_left);
  if (result == -1) {
    if (errno != EINTR) {
      // a real error happened,
      // so return an error result
    }
    // EINTR happened,
    // so do nothing and try again
    continue;
  }
  bytes_left -= result;
}
```

if first read only got n/3 bytes:
bytes_left = 2n/3

buf
⌊ want next read to
  start writing here

A. **buf**

B. **buf + bytes_left**

C. **buf + bytes_left - n**

D. **buf + n - bytes_left**

E. **We're lost…**

22

One method to `read()` n bytes

```c
int fd = open(filename, O_RDONLY);
char* buf = ...;  // buffer of appropriate size
int bytes_left = n;
int result;

while (bytes_left > 0) {
  result = read(fd, buf + (n - bytes_left), bytes_left);
  if (result == -1) {
    if (errno != EINTR) {
      // a real error happened, so return an error result
    }
    // EINTR happened, so do nothing and try again
    continue;
  } else if (result == 0) {
    // EOF reached, so stop reading
    break;
  }
  bytes_left -= result;
}

close(fd);
```

prevent infinite loop if EOF reached

readN.c   **23**

Other Low-Level Functions

❖ Read man pages to learn about:
- **write**() – write data
  - #include <unistd.h>
- **fsync**() – flush data to the underlying device
  - #include <unistd.h>
- **opendir**(), **readdir**(), **closedir**() – deal with directory listings
  - Make sure you read the section 3 version (*e.g.*, man 3 opendir)
  - #include <dirent.h>

❖ A useful shortcut sheet (from CMU):
http://www.cs.cmu.edu/~guna/15-123S11/Lectures/Lecture24.pdf

C Standard Library vs. POSIX

❖ C standard library implements a <u>subset of POSIX</u>

  ▪ *e.g.*, POSIX provides directory manipulation that C std lib doesn't

❖ C standard library implements <u>automatic buffering</u>

❖ C standard library has a <u>nicer API</u>

❖ The two are similar but C standard library builds on top of POSIX

  ▪ Choice between high-level and low-level

  ▪ Will depend on the requirements of your application

  ▪ You will explore this relationship in Exercise 4!

Extra Exercise #1

❖ Write a program that:
  ▪ Uses `argc`/`argv` to receive the name of a text file
  ▪ Reads the contents of the file a line at a time
  ▪ Parses each line, converting text into a `uint32_t`
  ▪ Builds an array of the parsed `uint32_t`'s
  ▪ Sorts the array
  ▪ Prints the sorted array to `stdout`

❖ Hint: use `man` to read about `getline`, `sscanf`, `realloc`, and `qsort`

```
bash$ cat in.txt
1213
3231
000005
52
bash$ ./extra1 in.txt
5
52
1213
3231
bash$
```

Extra Exercise #2

❖ Write a program that:

- Loops forever; in each loop:

  - Prompt the user to input a filename

  - Reads a filename from `stdin`

  - Opens and reads the file

```
00000000 50 4b 03 04 14 00 00 00 00 00 9c 45 26 3c f1 d5
00000010 68 95 25 1b 00 00 25 1b 00 00 0d 00 00 00 43 53
00000020 45 6c 6f 67 6f 2d 31 2e 70 6e 67 89 50 4e 47 0d
00000030 0a 1a 0a 00 00 00 0d 49 48 44 52 00 00 00 91 00
00000040 00 00 91 08 06 00 00 00 c3 d8 5a 23 00 00 00 09
00000050 70 48 59 73 00 00 0b 13 00 00 0b 13 01 00 9a 9c
00000060 18 00 00 0a 4f 69 43 43 50 50 68 6f 74 6f 73 68
00000070 6f 70 20 49 43 43 20 70 72 6f 66 69 6c 65 00 00
00000080 78 da 9d 53 67 54 53 e9 16 3d f7 de f4 42 4b 88
00000090 80 94 4b 6f 52 15 08 20 52 42 8b 80 14 91 26 2a
000000a0 21 09 10 4a 88 21 a1 d9 15 51 c1 11 45 45 04 1b
... etc ...
```

  - Prints its contents to `stdout` in the format shown:

❖ <u>Hints</u>:

- Use `man` to read about `fgets`

- Or, if you're more courageous, try `man 3 readline` to learn about `libreadline.a` and Google to learn how to link to it